OFLOPS: User manual

September 12, 2011

Chapter 1

Introduction

1.1 Why OFLOPS?

OFLOPS is an OpenFlow testing platform. The main focus of the platform is to provide a set of basic measurement test that allow developers to understand and quantify the capabilities of an OpenFlow enabled device and locate any possible bottleneck. This platform should be consider as a counterpart of the oftest platform. The oftest platformcan be used to verify the accurate implementation of the protocol in terms of semantics, while OFLOPS is focused on the performance aspect of the implementation.

In order to assess the performance of a network device that has both rich functionality and for which we have little idea of the implementation, we require a diverse set of concurrent measurements able to encompass all parameters of an experiment. Furthermore, to achieve high accuracy across a range of control and measurement tasks, we designed a unified platform that permits us to obtain information from multiple sources: data and control channels as well as SNMP to provide specific switch-state information.

A sample design of the platform is depicted in Figure **??**. The platform setups a single control channel with the switch and uses multiple network ports to generate traffic on the data plane of the switch. Additionally the platform supports SNMP protocol in order to read various MIB counters such as CPU utilization, packet counters etc.

Some of the features of the platform are the following:

- Modularity: The platform is split in 2 parts. There is the OFLOPS executable program which implements the core functionality of the platform and there is a set of dynamically loaded libraries that implement the required functionality for a specific test. In order for these two unit to communicate, the platform defines a rich event-driven API. Each dynamic test can implement a subset of the provided event handler and adjust the platform behavior to the needs of the test.

- Low overhead: The platform focuses to perform multi-level high precision

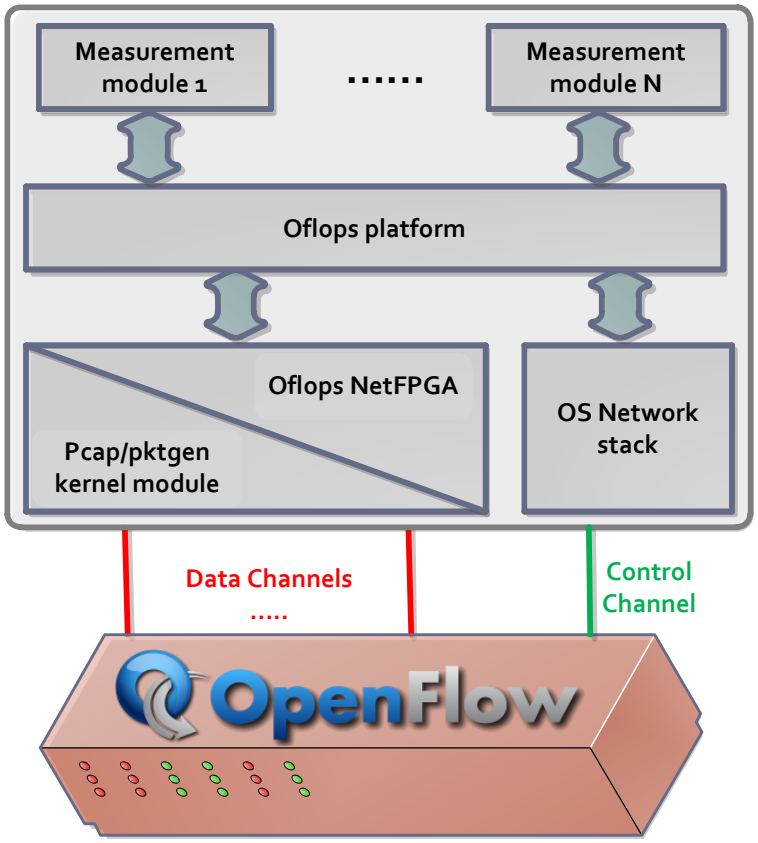

Figure 1.1: Oflops design schematic

measurements in order to benchmark the performance of the switch. In order to achieve minimum delay by the framework, special care has been given to reduce to a minimum any processing delay. In order to parallelize the processing and take advantage of modern multicore system, the platform is fully multithreaded.

- heterogeneity: The platform provides integration with multiple packet generation and capturing mechanisms. So far the platform can run both on commodity PCs with multiport NIC, as well as, NetFPGA equipped systems. Each platform provides different precision guarantees and costs.

Chapter 2

Installation

2.1 Source code

The source code of OFLOPS is available only through git repository. In order to get a copy of the code, users are required to have installed the git command line tools. the source code can be downloaded using the following command [1]:

```
git clone git://gitosis.stanford.edu/oflops.git
```

In order to build the code, OFLOPS requires two additional git packages. Firstly, it requires the source code of the NetFPGA packet generator c library. Because this library is also stored in a git repository, we have integrate the code as a git submodule and users can download it issuing the following command:

```
git submodule init && git submodule update
```

Secondly, users are required to have a version of the OpenFlow header file. By default the OFLOPS build configuration expects the OpenFlow reference implementation to reside in the directory where OFLOPS code. In order for a user to get access to the source code, he can issues the following commands:

```
git clone git://gitosis.stanford.edu/openflow
git checkout -b release/1.0.0 remotes/origin/release/1.0.0
```

For the compilation of the OFLOPS code it is required to install a series of libraries, namely pcap [2], GNU Scientific Library [3], libconfig [4] and net-snmp [5] Additionally, if the users require doxygen documentation of the source code, they can get automatically if they install the relevant package. The OFLOPS

[1] Currently, latest release of the source code is not on the git repo, but user can get an early access view of the code through an e-mail to cr409@cl.cam.ac.uk

[2] http://www.tcpdump.org/

[3] http://www.gnu.org/software/gsl/

[4] http://www.hyperrealm.com/libconfig/

[5] http://www.net-snmp.org/

build configuration script is configured to check for the presence of the doxygen software package and generate the appropriate Makefile if it is present. In the case of a CentOS 5.3 distribution all the required libraries can be installed using the following command.

```
yum install gcc automake autoconf libtool libpcap-devel net-snmp-
devel doxygen gsl-devel
wget http://www.hyperrealm.com/libconfig/libconfig-1.4.7.tar.gz
tar -xvzf libconfig-1.4.7.tar.gz
cd libconfig-1.4.7
./configure
make && make install
```

In order to build the source code readers must first build the c library of the NetFPGA OFLOPS packet generator. Assuming that the user current working directory is the OFLOPS directory, then the following sequence of commands should be issued:

```
cd netfpga-packet-generator-c-library/
./autogen.sh && ./configure && make
cd ..
./boot.sh && ./configure && make
```

In case the user wants to use an alternate location for the OpenFlow reference implementation, users may use the flag –with-openflow-src-dir= in the configuration script and define an alternative location for the default *'pwd'/../openflow*.

2.2 NetFPGA support

OFLOPS platform provides integration with the NetFPGA platform. We provide together with the OFLOPS source code a binary bitfile of a hardware design that enable hardware level packet generation and capturing. Both of these features are important for the precision of the measurement, because software packet generation and capture is, on one hand, limited in precision by the OS scheduler and, on the other hand, it is constraint by the bandwidth of the PCI bus. In order to use the hardware extension, the user needs to acquire and install a NetFPGA card and its base software system. We encourage readers to visit the NetFPGA website for further details [6]. Once the NetFPGA platform is configure on the host machine, users only need to download the appropriate bitfile to the FPGA circuit and modify accordingly the configuration file of OFLOPS. In order to download the bitfile, the following command is sufficient.

```
nf2_download netfpga-packet-generator-c-library/oflops_packet_generator.bit
```

[6] `http://www.netfpga.org`

Chapter 3

Running OFLOPS

3.1 Introduction

OFLOPS builds its testing modules as shared libraries. Modules are loaded at run time using the dlopen functionality of libc. In order to configure the test modules loaded during run-time and configure the parameters of a test, the OFLOPS tool uses a simple syntax language based on the libconfig format. In this chapter we describe the configuration parameter that the OFLOPS platform provides.

3.2 OFLOPS configuration files

The OFLOPS configuration file is based upon the libconfig file syntax, with a strict scheme for the configuration parameters. A set of example configuration files can be found in directory sample_config/. The OFLOPS configuration file consists of a unique group parameter, named OFLOPS, which contains a set of values that group the configuration parameters based on the generic aspect of the functionality that they define. Furthermore, the OFLOPS group contains a scalar parameters that defined which packet generation library will be used. The value is called *traffic_generation* and has 3 possible values:

- 1, use userspace packet generator.
- 2, user pktgen kenrel generator [1].
- 3, use NetFPGA-based packet generator.

Regarding the rest of the configuration parameters, they are grouped in the following group parameters:

[1] http://www.linuxfoundation.org/collaborate/workgroups/networking/pktgen

3.2.1 control channel configuration

For the control section of the configuration file OFLOPS parses the following parameter:

- *control_dev*: The name of the network device used for the control channel of the controller. This device shouldn't be used at the same time as a data channel.
- *control_port*: The TCP port on which the OFLOPS controller will listen on.
- *snmp_addr*: The IP address of the SNMP server of the OpenFlow switch.
- *cpu_mib*: A list of MIB entries for the CPU cores of the switch, semicolon separated.
- *in_mib*: The MIB entry of the packet counter of the input queue of the control channel switch port.
- *out_mib*: The MIB entry of the packet counter of the output queue of the control channel switch port.

3.2.2 data channel configuration

In order to configure the data channels used by an experiment, OFLOPS configuration file contains an array parameter called data which consist of variable number device description. The described network device will be used by the OFLOPS testing module in order to generate traffic on the data plane of the switch.

- *dev*: The name of the network device used by the channel. Each network device can only be used once per experiment.
- *port_num*: The ID of the port of the OpenFlow switch to which the network device is connected. This is required so that the testing modules will know which port ID to use on a flow modification message.
- *in_snmp_mib*: The SNMP MIB entry for the packet counter of the input queue of the switch port.
- *out_snmp_mib*: The SNMP MIB entry for the packet counter of the output queue of the switch port.
- *type*: The library used to capture packets over the specific data channel. The possible values are *nf2*, for NetFPGA packet capturing, and *pcap*, for libpcap-based packet capturing.

3.2.3 module configuration

Finally the configuration file contains an entry with a variable number of module configuration entries. The module entries can be used to locate and initialize appropriately a testing module. Specifically, for each entry OFLOPS accepts the following parameters.

- *path*: The path for the loadable library of the module.
- *param*: A space separate array of module parameters with their respective value. The list of parameters is specific for each module and is described in a great extend in Chapter **??**

3.2.4 Example configuration script

In order to provide an example of an OFLOPS configuration file we present the following configuration script. In this script we configure the control channel to use the device eth1 and open a tcp socket on port 6633, while it will query the snmp server on IP 192.168.1.2 in order get feedback from the switch. In this experiment we are configuring also two data channels on device nf2c0 and nf2c1 on which we will be using the the netfpga packet capturing functionality. Finally, for the experiment we dictate the OFLOPS to load the shared library file libopenflow_action_delay.so, to which we pass a series of parameters.

oflops.cfg

```
1  control: {
2              control_dev = "eth1";
3              control_port = 6633;
4              snmp_addr = "192.168.1.2";
5              cpu_mib="1.3.6.1.2.1.25.3.3.1.2.768;1.3.6.1.2.1.25.3.3.1.2.769";
6              in_mib="1.3.6.1.2.1.2.2.1.11.2";
7              out_mib="1.3.6.1.2.1.2.2.1.17.2";
8              snmp_community = "public";
9            };
10
11           data = ({
12                dev="nf2c0";
13                port_num=1;
14                in_snmp_mib="1.3.6.1.2.1.2.2.1.11.3";
15                out_snmp_mib="1.3.6.1.2.1.2.2.1.17.3";
16                type="nf2";
17                },{
18                dev="nf2c1";
19                port_num=2;
20                in_snmp_mib="1.3.6.1.2.1.2.2.1.11.4";
21                out_snmp_mib="1.3.6.1.2.1.2.2.1.17.4";
22                type="nf2";
```

```
23                  });
24
25              traffic_generator = 3;
26              dump_control_channel=0;
27
28
29  module: ({
30              path="libopenflow_action_delay.so";
31              param="flows=10 data_rate=10 pkt_size=150 ";
32              });
33               };
```

Chapter 4

OFLOPS Modules

4.1 Introduction

In this section we will provide an in depth description of the testing module that we have developed so far using the OFLOPS platform. For each module we describe the scenario it implements, the configuration parameters that it provides and its output information.

4.2 openflow_action_delay

This module is developed in order to measure the impact of the implementation of an action on the data plane of a switch. It has been observed in some OpenFlow implementation that some of the actions cannot be applied at line rate or they are not implemented at all. In order to test the switching performance of an action, the module initialized the switch flow table with a single flow that simply outputs packet to a specific output port. This action is kept in the cache for one minute, while the module generate matching traffic and measures the switching delay. After one minute the module sends a flow modification that adds on the action list of the flow a set of user defined actions, and modifies the output port while it continues to measure the switching delay incurred by the switch. By comparing how the number between the two phases, users can understand whether the action is implemented at line rate and whether the action incurs any significant delay.

The module provide the following set of parameters that the user can define:

- emphpkt_size: This parameter can be used to control the length of packets of the measurement probe, measured in bytes. Thus, together with the rate parameter, it allows indirectly to adjust the packet throughput of the experiment. (default 1500 bytes)
- *data_rate*: The rate of the measurement probe measured in Mbps. (default 10Mbps)

- *table*: The parameter define whether the inserted flow will be a wildcard(value of 1) or exact match(value of 0). (default 1)

- *action*: A comma separate string of entries of the format action_id/action_value. E.g. a value of 'b/1010,0/2' defines that the action will modify the tcp/udp port of the matching packet to a value of 1010 and the packet will be output on port 2. (default no action)

The module provide output for the result of the experiment on the log file. Specifically, it outputs 2 lines, one for each experiment phase, with the average, median and variance of the RTT delay of the probe and the packet loss, calculated through SNMP.

4.3 openflow_flow_stats & openflow_aggr_flow_stats

These modules are used to understand the possible impact of the flow stats extraction functionality of the OpenFlow protocol on the control and data plane of the switch. The two modules follow a similar experimental setup, but they differ on the type of the flow stat requests that they send. The first module uses the flow_stats_request OpenFlow message type, while the latter uses the aggr_flow_stats_request. The scenario for this experiment is as follows: the flow table is initialized with a set of flows for which OFLOPS generates relevant traffic. The measurement traffic consists of 2 probes: a *constant probe*, that matches a single flow, and a *variable probe*, that uses at random any of the inserted flows. The experiment runs for 1 minute and at the same time the controller queries the switch for its flow statistics over specific intervals. The module stores statistics for the measurement probe, as well as, the reply time for the flow statistics and the CPU utilization.

The module provides the following configuration parameters:

- *flows*: The total number of unique flow that the module will initialize the flow table of the switch. (default 128)

- *query*: The number of unique flows that the module will query the switch in each flow request. Because the matching method of the module is based on the netmask field of the matching field. (default 128)

- *pkt_size*: This parameter can be used to control the length of the packets of the measurement probe. It allows indirectly to adjust the packet throughput of the experiment.

- *data_rate*: The rate, in Mbps, of the variable probe. (default 10Mbps)

- *probe_rate*: The rate, in Mbps, of the constant probe. (default 10Mbps)

- *query_delay*: The delay, in microseconds, between the different stats requests. (default 10000 usec)

- *print*: A parameter that defines whether the module will output full per packet details of the measurement probes. If this value is set to 1, then the module will print on a file called "measure.log" for each capture packet a comma separated record with the timestamps of the generation and capture times of the packet, the packet id, the port at which the packet was captured and the flow id of the flow that was used in order to switch the packet. (default 0)

- *table*: This parameter controls whether the inserted flow will be a wildcard(value of 1) or exact match(value of 0). (default 0)

The output of the module is printed in the log file. The module prints, after the end of the experiment, a log line for each flow_stat_reply containing timing information. Additionally, the module prints RTT and loss statistics for each measurement probe. Finally, the module queries the switch every 10 seconds and outputs CPU information in the log file.

As a future development of this module we are considering the potentiality to estimate the accuracy of the statistic extraction mechanism. Unfortunately though, so far we are unable to synchronize accurately the result of packet counters of the capturing library, with the results of flow_stats_reply packets.

4.4 openflow_packet_out

This module is developed to measure the performance of the packet_out functionality of the OpenFlow protocol. This message type, allows a controller to inject packets in the network by explicitly sending the packet data to the switch and defining the output port of the packet. The module provide basic measurement on the performance of the OpenFlow implementation given a set of parameters. During the experiment, a constant flow of packet_out message is sent to the switch over the control channel while the module capture output packets on the data plane of the switch. This way the module can calculate the processing delay incurred to such message types, as well as, any implication that may occur when the controller increases it's sending rate.

The module provides the following configuration parameters:

- *pkt_size*: This parameter can be used to control the length of the packets of the packet_out message in bytes. It allows indirectly to adjust the packet throughput of the experiment. (default 1500 bytes)

- *probe_snd_interval*: This parameter controls the data rate of the measurement probe, in Mbps. (default 10Mbps)

- *print*: This parameter enables the measurement module to print extended per packet measurement information. The information is printed in log file. (default 0)

The module outputs in the log file of the program SNMP information from the switch, queried every 10 seconds, as well as statistics for the RTT and packet

loss of the packets inserted using the packet_out message type. If the parameter 1 is set to one, then the module will additionally print per packet timestamps.

4.5 openflow_packet_in

This module is developed to measure the performance of the packet_in functionality of the OpenFlow protocol. This message type is generated when a packet is received by the switch, that does not match any flow in the flow table. For this experiment, the module initially removes any flows in the flow table and starts to generate traffic at a specific rate on one of the ports of the switch. In parallel, the module receive packet_in messages and computer per packet the processing delay.

The module provides the following configuration parameters:

- *pkt_size*: This parameter can be used to control the length of the packets of the measurement probe. It allows indirectly to adjust the packet throughput of the experiment. (default 1500 bytes)
- *probe_snd_interval*: This parameter controls the data rate of the measurement probe, in Mbps. (default 10Mbps)
- *print*: This parameter defines if the measurement module prints extended per packet measurement information. The information is printed in log file.

The module outputs in the OFLOPS log file SNMP information from the switch, queried every 10 seconds, as well as statistics for the RTT and packet loss of the measurement probe, as measured through the packet_in messages.

4.6 openflow_reactive

This module can be used to understand the scalability properties, when an OpenFlow switch is used in a reactive way, similar to the functionality that the NOX controller provides. In this case, the controller inserts flows only when a packet_in event is generated. In order to simulate a usage scenario like this, the module initially removes any flows from the switch flow table and starts to generate measurement traffic. The measurement traffic matches a specific number of exact match flows and packets are generate in a way that matching flows are used sequentially. The module logs for each flow the time it received the first packet, the time it received the packet_in message and the time a packet of the flow was received on the data plane.

The module exports the following parameters:

- *pkt_size*: This parameter can be used to control the length of the packets of the measurement probe. It allows indirectly to adjust the packet throughput of the experiment. The parameter uses bytes as measurement unit.

- *probe_rate*: The rate of the measurement probe, measured in Mbps.
- *flows*: The number of unique flows that the measurement flows will generate.
- *print*: This parameter enables the measurement module to print extended per flow measurement information. The information is printed in log file.

The module outputs in the OFLOPS log file the total delay incurred in order to setup all of the flows of the experiment. In addition, the module polls the switch for SNMP packet counters and CPU usage every 10 sec and when the experiment completes and save the query result in the log file. Finally, if the print parameter is enabled, the module output in a CVS files a set of records, one for each flow, that contain: the flow id and the timestamps of the time the first packet of the flow was generated, the time when the first packet_in event of the flow was received by the controller and the time the first packet of the received on the data plane.

4.7 openflow_add_flow & openflow_mod_flow

This set of modules can be used to measure the scalability properties of the flow table. The modules measure the delay that the switch incurs when the controller modifies the content of the flow table. The difference between the two modules is that the first module considers only flows insertions, while the later focuses on flow updates. The experimental scenario works as follows: During the initialization of the experiment a set of flows is inserted in the flow table in order to switch all of the traffic in the data plane. For the case of flow additions, the module insert a single wildcarded flow that outputs data to port 3 of the experiment, while for flow modification, the module inserts a number of flows in the flow table equal to the number of flows in the data plane that output matching packets to port 3. The measurement traffic consists of 2 probes: a *constant probe*, that matches a single parameter, and a *sequential probe*, that uses in a round Robbin manner all of the inserted flows. We use two measurement probes because we are interested to measure the delay to insert both a single flows as well all of the flow. The experiment keeps the initial content of the flow table for 30 seconds, in order for the switch to reach a steady state, and then sends a series of flow modification and a barrier request. The module monitors all ports and stores information of the time when each new flow modification becomes active and when the barrier reply is received.

The module provides the following configuration parameters:

- *pkt_size*: This parameter can be used to control the length of the packets of the measurement probe. It allows indirectly to adjust the packet throughput of the experiment. The parameter uses bytes as measurement unit. The parameter applies for both measurement probes.
- *probe_rate*: The rate of the sequential probe, measured in Mbps.

- *data_rate*: The rate of the constant probe, measured in Mbps.
- *flows*: The number of unique flows that the module will update/insert.
- *table*: This parameter controls whether the inserted flow will be a wildcard(value of 1) or exact match(value of 0). For the wildcard flows, the module wildcards all of the fields except the destination IP address.
- *print*: This parameter enables the measurement module to print extended per packet measurement information. The information is printed in a separate CSV file, named "action_aggregate.log".

The module logs in the OFLOPS log file the timestamps when the flow modification and barrier request transmission starts(entry label: START_FLOW_MOD, BARRIER_REPLY) and ends(entry label: END_FLOW_MOD) and when each flow becomes active in the data plane(entry label: FLOW_INSERTED). Additionally, the module prints for each channel statistics regarding the RTT and packet loss for each data channel.

4.8 openflow_echo_delay

This module can be used to measure the processing delay of the control channel. Specifically, it generated echo_request messages for one minute, and calculates the delay for the switch to reply. The module provides a single configuration parameter, *delay*, which define the inter-request delay in microseconds.

The module outputs in the log file a line for each echo_reply, that contains all the required timing information, for offline analysis. The module also output at the end of the measurement an entry with mean, median and variance of the reply delay as well as any possible reply loss.

4.9 openflow_interaction_test

This test was developed in order to understand how the flow stats extraction mechanism may interfere with other control plane operations. In this experiment we repeat the scenario described in the openflow_flow_stats module, but after 1 minute the module tries to modify the flow table of the switch. In parallel the module monitors the time it takes for the new flows to become active on the data plane.

The module exports the following configuration parameters:

- *pkt_size*: This parameter can be used to control the length of the packets of the measurement probe. It allows indirectly to adjust the packet throughput of the experiment. The parameter uses bytes as measurement unit. The parameter applies for both measurement probes.
- *probe_rate*: The rate of the sequential probe, measured in Mbps.

- *data_rate*: The rate of the constant probe, measured in Mbps.
- *flows*: The number of unique flows that the module will update/insert.
- *table*: This parameter controls whether the inserted flow will be a wildcard(value of 1) or exact match(value of 0). For the wildcard flows, the module wildcards all of the fields except the destination IP address.

The module will output on the OFLOPS log file, all the details described in the openflow_flow_stats and openflow_add_flow modules.

4.10 openflow_timer

With this measurement module, users can quantify the precision of the internal timers of an OpenFlow switch. The OpenFlow protocol, provides for each flow a couple of parameters that allows the controller the time for which a flow will be effective. Once the timers have expired, the switch is obliged to remove the flow from the flow table and inform the controller with an appropriate message, if the flow has a specific flag set. In this experiment, the flow table is initialized with a default flow with low priority that switches packets to port 2 and in parallel it generate a sequential measurement probe that match a specific range of flows. After a second of experimental run time, the controller insert a set of new high priority flows that match the packets of the measurement probe and output packets to port 3 of the switch. Each flow has a specific hard timeout value assigned to it as well as the OFPFF_SEND_FLOW_REM flag set. The module monitors the duration that the flows are active, using the data plane of the switch, as well as the duration feedback that it receives from the FLOW_REM message and stores them in the log file of OFLOPS.

The module provides the following configuration parameters:

- *pkt_size*: This parameter can be used to control the length of the packets of the measurement probe. It allows indirectly to adjust the packet throughput of the experiment. The parameter uses bytes as measurement unit. The parameter applies for both measurement probes.
- *data_rate*: The rate of the constant probe, measured in Mbps.
- *flows*: The number of unique flows that the module will update.

4.11 snmp_queue_delay

This module is developed in order to measure the switching performance in the data plane of the switch. This test is important mostly for software switches, as hardware switches perform switching at line rate using specialized hardware designs resulting in negligible processing delay. In this experiment the module initializes the flow table with a single flow that output packets to a specific port. The module generates traffic at various rates for a specific time period

and measure for each packet the switching delay. Specifically, the module tests for data rates of 1, 10, 64, 128, 256, 512 and 1000 Mbps.

The module accepts the following configuration parameters:

- *pkt_size*: This parameter can be used to control the length of the packets of the measurement probe. It allows indirectly to adjust the packet throughput of the experiment. The parameter uses bytes as measurement unit.
- *duration*: The duration of the measurement period for each probing rate. The measurement unit is seconds and the default value is 30.

The module will output in the OFLOPS log a set of records which provide for each probing rate the mean, median and variance of the RTT and the packet loss.

4.12 snmp_tuple_test

In this module we are testing all possible combinations of the OpenFlow toppled in order to quantify any possible the impact of the tupple format to the data plane. This test design occurred by the fact that some of the early implementations, did not support at full extend the OpenFlow tupple. The module uses a single flow entry and a single flow measurement probe. For each possible tupple combination the controller send a flow modification with the appropriate flow match and then sends traffic for a specific period of time. For each test, the module stores in the log file the RTT and loss statistics of the flow.

The module provides two possible parameters for this module: *pkt_size* and *duration*. The pkt_size controls the length of the packet of the measurement probe. The duration is used to define to the module the duration of each tupple test. Currently the default value is 10 seconds. The module outputs on the OFLOPS log file for each match case a line with statistics for the RTT and packet loss of the measurement probe.

4.13 openflow_dummy

This module is a simple reference module that is included in order to exemplify to user all the potential API calls that a module can implement. This module does not implement any functionality and should not be used in any test.

Chapter 5

Developing OFLOPS modules

5.1 OFLOPS API design

OFLOPS is an event driven modular platform. Its main focus is to allow rapid development of Openflow usage scenarios. In order to achieve this, the code is split in two parts: The main platform and the modules. The main platform is repsonsible to implement a basic functionality for an openflow controller. It implements code that initializes a control channel with the switch, an asynchronous SNMP client, a simplistic traffic generator and a high precision traffic generator. On the other hand, the module contain all the scenari specific code. The platform is responsible to abstract to a higher level the OpenFlow interaction and can tae advantage of this functionality in order to implement with little code complex experiments. In this section I will cover the details on how to develop an OFLOPS module.

5.2 Module life-cycle

The design that we have chosen for the OFLOPS module functionality is build around dynamic linking. Each module is compiled as a run time loadable library. Each module, in order to permit to the main platform to push event, must implement specific functions in its code. Each OFLOPS module can be seen as a set of implementations of specific function signatures, along with a scenario specific internal module state.

The module lifecycle consist of 4 different phases, over which the main platform will interact with the module using different methods: 1) *Bootstrap*: In this phase OFLOPS will call the init function of the module and pass as a argument the initialization string that the user provides throught the configuration file. The module is responsible to allocate and initiliaze all the internal variables of

the module. 2) *Intialization*: In this phase the OFLOPS platform has setup a control channel with the switch but hasn't start to interact with the switch and run the various threads of the module. In this phase the module is expected to setup any required initialization of the switch flow table and schedule the various events of the module. 3) *Run*: In this phase the platform will run the function that start the packet generation mechanism and call all of the implemented openflow and SNMP event callbacks. A detailed analysis of the various events of this phase is described in Section **??** 4*Termination*: In this phase the module has stopped running the main event loop. The platform will call the destroy function of the module, where it is expected for the module to execute any result processing code and release all allocated memory.

5.3 OFLOPS API and Events

5.3.1 Control Channel Events

In order to handle singalling on the control channel the OFLOPS platform uses a separate thread to monitor the control channel, parse received packet and convert them to appropriate events for the modules. In order for the modules to receive data from the control channel, they should implement a specific function, depending on the openflow event that they are interested to get informed. The set for available handlers that the OFLOPS platform support is the following:

Listing:

```
1  /* handle packet_in events */
2  int of_event_packet_in(struct oflops_context *ctx,
3    const struct ofp_packet_in * pktin);
4
5  /* handle flow_removed events */
6  int of_event_flow_removed(struct oflops_context *ctx,
7    const struct ofp_flow_removed * ofph);
8
9  /* handle openflow echo requests events */
10 int of_event_echo_request(struct oflops_context *ctx,
11   const struct ofp_header * ofph);
12
13 /* receive port_status events */
14 int of_event_port_status(struct oflops_context *ctx,
15   const struct ofp_port_status * ofph);
16
17 /* handle all remaining openflow events */
18 int of_event_other(struct oflops_context *ctx,
19   const struct ofp_header * ofph);
```

Each function signature consists of the data context of the module along with a pointer to the data of the packet. OFLOPS also permits to module

to inject custom packets to the control channel. In order to achieve this the plaform provides the following functions:

Listing:

```
1 size_t oflops_send_of_mesgs(struct oflops_context *ctx,
2   char * buf, size_t buflen);
3 int oflops_send_of_mesg(struct oflops_context *ctx,
4   struct ofp_header * hdr);
```

This set of functions can be used only once the control channel has been established. oflops_send_of_mesg sends only a single openflow packet and infers the size of the packet buffer, through the length field of the openflow header. oflops_send_of_mesgs permits the developer to send a number of packets, but requires to define the size of the packet buffer passes as an argument. If the developer wishes to atain higher control of the tcp socket of the control channel, it can use the control_fd field of data context of the module, but he should be aware that the socket is non blocking and aggressive writes on the socket may lead to packet losses.

5.3.2 Data Channel Events

As it has already been discussed previously in this document, the OFLOPS platform contains a basic simple packet generation mechanism. The platform currently provides 3 possible alternatives: a user space packet generator, a kernel space packet generator that use the pktgen module or a NetFPGA based packet generator. The selection of the pacet generation mechanism is done though the configuration file and is abstracted from the module code. In order to bootstrap the packet generation mechanism the module must provide an implementation of the following function:

Listing:

```
1 int handle_traffic_generation(struct oflops_context * ctx);
```

In this function the developer can initialize the packet generation mechanism. The platform provides the following functions to developers:

Listing:

```
1
2 /* called before any interaction with the packet
3   generator */
4 int init_traf_gen(struct oflops_context *ctx);
5
6 struct traf_gen_det {
7   char src_ip[20];
8   dst_ip_max[20], dst_ip_min[20];
9   char mac_dst[20], mac_src[20];
```

```
10    uint16_t udp_src_port, udp_dst_port;
11    uint32_t pkt_size;
12    uint16_t vlan;
13    uint16_t vlan_p;
14    uint16_t vlan_cfi;
15    uint32_t delay;
16    uint64_t pkt_count;
17
18    /* RANDOM -> packet generated uniformly at
19    random for the specified ranges
20     "" -> packets are ganerated sequentially
21     iterating all possible pairs */
22    char flags[1024];
23  };
24
25  /* Adding a packet generation description to
26    on a specific data channel */
27  int add_traffic_generator(struct oflops_context *ctx,
28    int channel, struct traf_gen_det *det);
29
30  /* start packet generation*/
31  int start_traffic_generator();
32
33  /* terminate packet generation */
34  int stop_traffic_generator( oflops_context *ctx);
35
36  /* returns a string with detailed informations
37     regrading packet generatio process */
38  char *report_traffic_generator(oflops_context *ctx);
39
40  struct pktgen_hdr {
41    uint32_t magic;
42    uint32_t seq_num;
43    uint32_t tv_sec;
44    uint32_t tv_usec;
45  };
46
47  /* Extract packet generation details from a packet
48    */
49  struct pktgen_hdr *extract_pktgen_pkt(oflops_context *ctx, int port,
50                                        unsigned char *b, int len, struct flow
```

In order to initialize the packet generation process the developer must use the init_traf_gen function before ay other calls to the packet generation system. Once the mechanism has beed initiated, the module can add packet generation descriptions to each data channel. A packet generation description is provided

throught the traf_gen_det structure which defines the way packets are generated. Developers currently are only able to generate packet that variate based on the destination address. The fields of the packet

5.3.3 Timer Events

The main OFLOPS platform provide a simple mechanism that allows module to construct custom timed events. The main function to register an event is throught the function:

Listing:

```
1 int oflops_schedule_timer_event(struct oflops_context *ctx,
2 struct timeval *tv, void * arg);
```

where ctx is the data context of the module, tv is a timeval structure that define after how much time should the timer invoke the timer handling function, and arg is a module defined object that will be pushed to the event handling function when the event fires. In order for the module to receive timer events, it must implement the following function:

Listing:

```
1 int   handle_timer_event (struct oflops_context *ctx,
2   struct timer_event *te);
```

In this function the module can define in a custom way the various events. The details of the event are contained in the structure timer_event. The structure has the following format:

Listing:

```
1 typedef struct timer_event
2 {
3         int timer_id;
4         void * arg;
5         struct timeval sched_time;
6 }
```

In order for the code to detect the event fired each time, code developers can use the arg field of the event in order to differentiate the different events.

5.3.4 SNMP events

As part of the main OpenFlow platform, module are able to query asynchronously the SNMP service of the switch. The SNMP functionlity is similarly event driver as the rest of the functionality of OFLOPS. Module can send queries using the function:

Listing:

```
1 int oflops_snmp_get(struct oflops_context * ctx,
2   oid query[], size_t len);
```

The user can set in each call a query for a single OID. This is defined though the parameters query and len. Once a query has been send, a separate thread will listen on a UDP socket, receive the reply and generate an OFLOPS event to the module. In order for a module to capture SNMP event through an OFLOPS

Listing:

```
1 int handle_snmp_event(struct oflops_context * ctx,
2   struct snmp_event * se);
```

The snmp_event object, passed as an argument to the callback has the following format:

Listing:

```
1 typedef struct snmp_event
2 {
3   /** SNMP variable list
4    */
5   struct variable_list* reply;
6   /** Pointer to SNMP reply
7    */
8   struct snmp_pdu* pdu;
9 }
```

The data of the reply can be found in struct snmp_pdu. In order to parse the information programmers should use the API provided by the net-snmp library [1].

[1] http://www.net-snmp.org/wiki/index.php/TUT:Simple_Application